Applying Marea Middleware to UAS Communications

Juan López, Pablo Royo, Cristina Barrado and Enric Pastor
Department of Computer Architecture, Technical University of Catalonia (UPC)
08860 Castelldefels (Barcelona), Spain

Unmanned Aircraft Systems (UAS) are increasing their communication subsystems complexity as they are using several data links both for redundancy and for tacking with different communication requirements: different ranges, bandwidth, power consumption or cost. Inside the airframe a similar tendency with respect to communications is appearing. Modern digital avionics are mainly implemented as distributed computing architectures. This work describes Marea, a middleware specifically designed to fulfill UAS communications and their application to the design of complex distributed UAS avionics.

Nomenclature

UAS = Unmanned Aircraft System
IMA = Integrated Modular Avionics
Marea = Middleware Architecture for Remote Embedded Applications
USAL = UAS Service Abstraction Layer
ARQ = Automatic Repeat reQuest
RTOS = Real Time Operating System
VAS = Virtual Autopilot System
COTS = Commercial-Off-The-Shelf

I. Introduction

Unmanned Aircraft Systems (UAS) usually use several communication links both for redundancy and for tacking with different communication requirements: different ranges, bandwidth, power consumption or cost. Control data, telemetry and mission data, mainly imagery and sensor data, are the most common payload of these links. In most cases dedicated raw RF links are used, however packet based networking is more flexible and allow most complex payloads and protocols. As UAS missions support more intelligent and cooperative behavior, UAS communication systems will benefit from more complex communication patterns like request-response, asynchronous events or remote procedure calls.

Inside the airframe a similar tendency with respect to communications is appearing. Modern digital avionics are mainly implemented as distributed computing architectures. Two different approaches are given: federated and modular. In federated avionics, every avionics functionality is integrated into a black-box and none resource is shared between avionics systems other than the communication buses. In the IMA approach, functionalities are distributed into logical partitions which may be allocated in the same physical computing module or into a different one[12]. This resource sharing involves weight and power savings since resources can be used more efficiently. In both cases, communication infrastructure is an important part of the system.

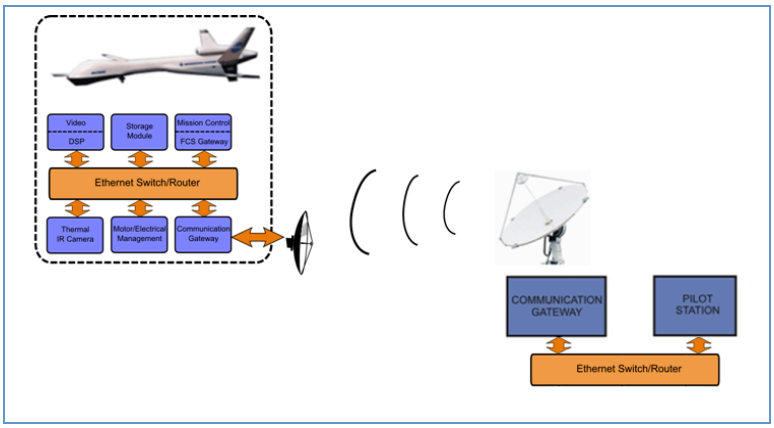

Figure 1. UAS Communication Architecture. *The UAS seen as a network of distributed components.*

UAS avionics, especially in small UAS, are usually of less complexity than not the present on airliners… less instrumentation, less engines, no need to monitor and control the pressurization, etc. However, on the other hand, in real autonomous UAS, the onboard avionics should control the flight, navigation, mission and payload of the aircraft. This involves more complex software as it should implement "intelligent" or at least autonomous behavior. Most of this intelligence is implemented as complex interactions between different UAS avionics components.

This way, communications both inside the UAS airframe and between the UAS and the ground control station are gaining momentum and significance. Transparency and flexibility are important in these communication mechanisms.

The remainder of this paper is organized as follows: Section 2 outlines the Marea middleware, their communication primitives and its internal architecture. Section 3 describes the Communication Gateway, a configurable subsystem that allows connecting the Marea enabled system onboard the aircrafts with the systems on ground. And finally Section 5 presents concluding remarks and outlines future work.

II. Marea Middleware

Middleware-based software systems consist of a network of cooperating components which implement the logic of the system and an integrating middleware layer that abstracts the execution environment and implements common functionalities and communication channels. In this view, the different components are semantic units that behave as producers of data and as consumers of data coming from other components. The localization of the other components is not important because the middleware manages their discovery. The middleware also handles all the transfer chores: message addressing, data marshaling and demarshalling delivery, flow control, retries, etc.

Our Marea middleware proposes a modular architecture based on services[6]. The avionic system is composed of set of distributed elements, known as services, which operate on top of the middleware communication framework. The services are collocated over the different computational nodes that are connected by a low-cost local network. This interconnection scheme is very flexible and cost-effective.

Most of the present middleware is focused on client-server relationships between components[5,7,8]. While this approach is applicable for most systems, it does not take benefit of the multicast capabilities of local networks. Publish-Subscribe communications are better suited for this sort of systems. Marea combines both client-server and multi-point message communications and it is specially suited to communicate low-cost components interconnected by local networks.

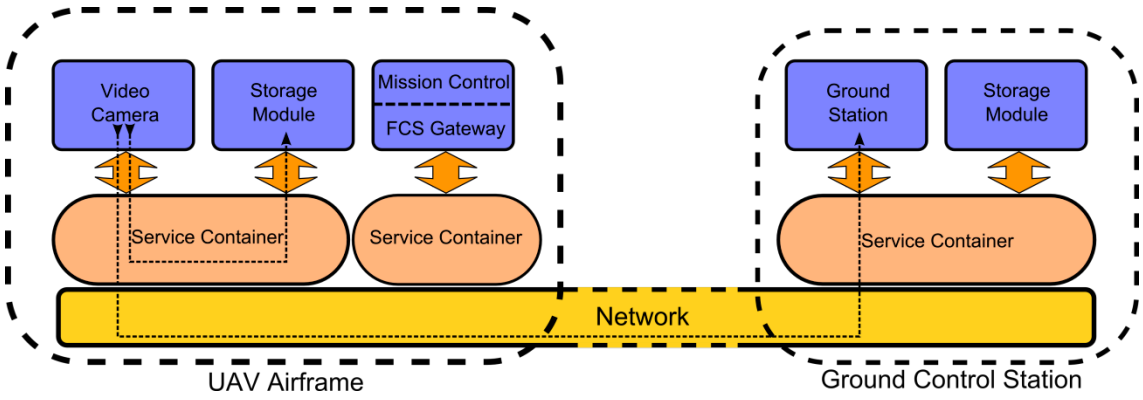

Figure 2. Marea middleware. *The figure shows two Marea containers supporting several services both onboard the UAS and on the Ground Control Station.*

A. Avionics services communication

In our architecture, the middleware takes the form of a component called service container. The services are executed and managed by a service container that is unique in each node of the distributed system network. The service container can manage several services and provides common functionalities (network access, local message delivery, name resolution and caching, etc.) to the services it contains.

The communication primitives that Marea provides to the services are capable to transparently locate and attach to the provider services with no need of knowing the final physical location of them. Four communication primitives (Variable, Event, Remote Invocation and File Transmission) give the avionics developer a wide field of possibilities to interconnect and to make interact services.

A Variable is a structured, and generally short, information offered by one service in a publish-subscribe way. This information may be sent at regular intervals or when changes occur. An Event is similar to a Variable but the middleware guarantees the reliability of the transmission. Events are used to inform of occasional or important facts to other services. Remote Invocation is the classical way to model interactions between distributed components. It mimics a procedure call in non distributed environment. Finally a File Transmission is a data transfer of continuous information. This includes photography images, video, configuration files or even program code.

A Marea communication primitive identifies exchanged data rather than their providers or consumers. This functioning principle is similar to actual avionics buses such as ARINC 429. This bus broadcasts transmitted data, with an extra information (label), to all linked equipment and only the ones who have recognized the label use data. In that sense, Marea like ARINC 429[10] or APEX[11] ports allows allow to link producers and receivers of data, without a priori knowledge of the physical location of them.

B. System architecture

Marea has been designed with interoperability and flexibility in mind. Their internals have been distributed in several layers allowing the addition of new components such as transports and codifications. Marea has its components distributed in four layers: Protocol, Encoding, Presentation and Transport. This distribution loosely resembles the PEPt architecture from Sun[2].

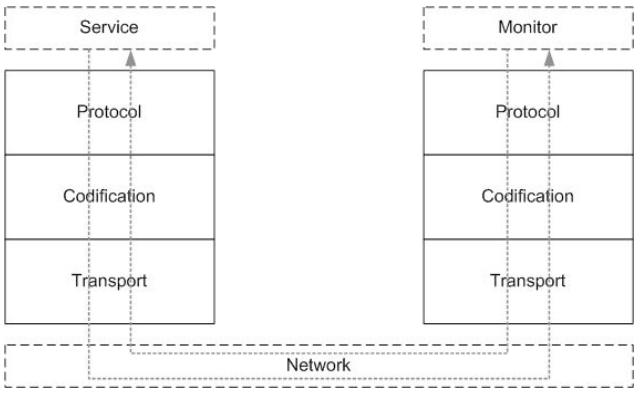

Figure 3. A high level view of Marea middleware layers. *The figure shows two Marea containers with their internals layers: Protocol, Encoding and Transport.*

The transport layer is the lowest level layer on the Marea stack. It is on charge of sending and receiving data from the underlying network and to interface to the upper Encoding layer. The Marea middleware is message based so the transport layer is especially well suited for datagram based networking (i.e UDP). In the case of stream based communications, the transport layer implementation has two options: a) one stream can be opened for each message, and after its sending, the stream is closed. b) as the stream initialization usually require a long handshaking phase it is better for performance to reuse the established connections, waiting for new messages to be send to the same destination through a unique stream. In its current status, Marea offers an UDPTransport, TCPTransport and PersistentTCPTransport. They implement basic UDP and TCP transports for Marea messages.

In the case of PersistentTCPTransport as the first connection with other node is established it is cached in a table indexed by node. All the following messages between the nodes that have established the connection are sent through this connection. For this transport, a specific transport has been designed to dissect the stream into messages. Basically, each message is preceded by a header that indicates the length of the next message. This way, the receiving transport layer knows when a message is complete and it can pass it to the upper Encoding layer. The protocol is similar to BEEP[3] in the sense that multiplexes datagrams or messages into a TCP stream however our implementation is simpler and not text based.

Marea transport layer is not limited to IP based communication. Although the most common connection between Marea containers is IP over Ethernet or Wi-Fi other simpler networks are possible, for example serial RF modems. Each of these connections are represented in Marea by an ITransport interface. This interface provides access to the underlying device and allows to send and receive data and to get link status information (i.e. link quality, expected bandwidth and latency). This modular approach allows to easily add new datalinks to the system.

The encoding layer is on charge of transforming Marea messages into byte arrays and vice versa. Data received in the transport layer is cut into datagrams and sent to the upper encoding layer. Here, the received bytes are translated into a message that can be processed by the upper layers. In a similar way, when the Service container needs to communicate with other Service containers it forwards a Message to the encoding layer that transforms it into a coded byte array. Marea provides several coders and decoders to allow easy interoperability and devices/network adaptability. Depending on the selected encoding, the balance between codification latency and network bandwidth needed to transmit the encoded message can be modified. Currently, Marea supports one optimized binary encoder and one text-readable XML encoder.

Finally, the Protocol layer is on charge of scheduling the message depending on its priority and after that, process it. Control messages managing the discovery, publication and subscription of services are managed on this layer, while data messages are feed to the subscribed services. All the services provide a common interface to get notifications and data from the container and other services.

The service interface contains operations to manage the following operations:

- Start(): This operation is invoked for all the initial services at container startup. A service should respond to this call by allocating its needed resources and start performing its functionality.
- Stop(): A service is asked to finish by receiving this call. Container issues this call at system shutdown, reconfiguration or during problem containment.

- Run(): Usually, a service requires a thread to perform its job. The Run() function contains the code to be executed by this main thread.
- VariableChanged(): When a subscribed Variable primitive gets new data, the container calls this function in the subscribed services. The function acts as a callback returning the originating service and the sent data.
- EventFired(): This is the equivalent callback but for Event primitives.
- FunctionCall(): Finally, Function primitives are notified by issuing this call in the producer service. The service executes the indicated operation and returns the results to the container, which will send them to the calling service.

The service container will issue this operations on the different services, as they are initiated, stopped or they receive new data from others services. In addition, the service container will contact with other containers in the local network to discover new services and manage remote publications and subscriptions. More details about the protocol between containers are given in the next subsection.

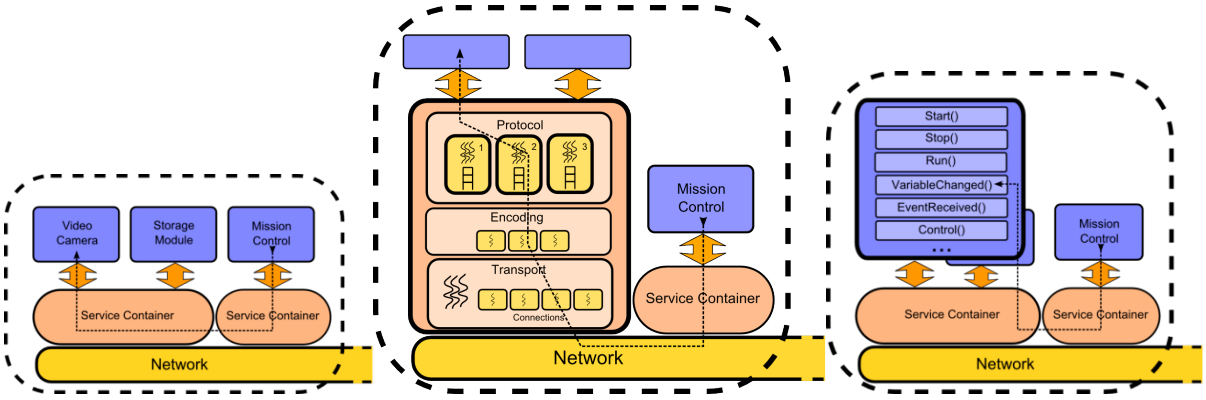

Figure 4. Marea Internal Implementation. *The figure shows the internal delivery of data between components on the same local network.*

Figure 4 shows three different views of a Marea service container receiving and processing a message through its different layers. In the leftmost figure, we can see two service containers deployed over the same aircraft internal network. More specifically, the Mission Control service notifies the Video Camera service to take an aerial photo by using a Variable communication primitive.

In the central figure, we focus on what happens inside the service container and its layers. First, we found the Transport layer where the different transports keep the communication with other service containers. Next, we found the Encoding layer where the raw data received in a transport is decoded to be scheduled and processed in the upper Protocol layer.

In the rightmost figure, the Mission Control notifies the Video Camera by using a Variable primitive so the VariableChanged() callback is called for all the subscribed services in the container. Then, each service will react to the input according its functionality.

C. Protocol

In this section we are going to describe the protocol that Marea uses to communicate two different service containers. It is important to notice that the services does not communicate directly with other services, they always rely in the container to perform their communication. This allows the container to apply some optimizations and to reuse the communication flows between service containers. For example, imagine that a service container runs a service that is required by two other services contained in a remote container. When the producer service generates a new data sample, both consumers should receive the new data. Marea containers keep information about publishers and consumers and only they establish one publish-subscribe relationship while sending only one data flow between containers. The consumer service container will replicate the data flow for its two consumer services.

In addition, the Marea protocol has been designed to take profit of the multicast capabilities of local networks, for example Ethernet. Under standard load, an Ethernet network can deliver UDP datagrams to several nodes in

multicast with almost no packet losses. Unfortunately, under heavy loads, this is not guaranteed and mechanisms like Automatic Repeat reQuest (ARQ) and control flow mechanism should be implemented. TCP streams are the typical implementation of these mechanisms. While different service communications have different requirements and constraints regarding its reliability and performance, Marea provides different communication primitives and underlying protocols to be easily adaptable.

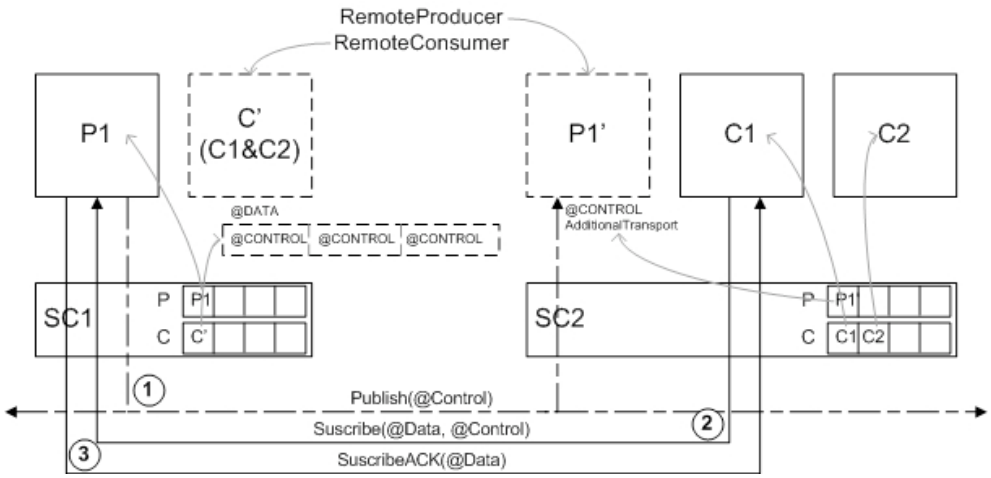

Figure 5. Marea protocol. *The figure shows two Marea containers establishing a publisher-subscriber relationship.*

The Marea protocol is divided in three phases: publish-subscribe establishment, data transfer and publish-subscribe relationship termination. Figure 5 shows schematically all the steps involved in the first two phases:

1. The life cycle of a publish-subscribe primitive, such as a variable, starts with the publication of its location to all the nodes of the network. This message is broadcasted in UDP and contains the control transport address of the service container that offers the service. Although a service container can have several transport modules listening on different ports and interfaces for both data and control messages, it chooses one TCP control transport that identifies univocally the service container in the marea network. In the figure 5 we can see as the service container 1 (SC1) informs of the availability of the service P1 to the network and its control address.

2. When another service container (SC2) receives this message it stores the availability of the service in its internal structures. For this, it creates a proxy service that represents the service managed by the other service container (P1' in the figure). This proxy contains the control address of container that manages the original service and it is added to the publisher list of the container as a normal service.

3. When a new service wants to consume this primitive published in a remote container, in reality it only sees the proxy service. When the proxy service notices that some service has requested its services it sends a subscribe message to the original control address. In this message the proxy sends the control address of the requesting container and a suggested data address where it wants to receive the incoming data of the primitive. The suggested data transport can be a unique address or a prioritized list from where the provider container can choose. Depending on the QoS required for the data, the proxy can ask for a TCP address or a broadcast UDP address.

4. The provider container receives the remote subscription with the list of suggested data address. In this step the container can negotiate the address to use, for example if it not support multipart it can choose a standard UDP or TCP transport. Finally, it sends a SubscribeACK with the chosen transport address to the subscriber container.

5. Similarly to the consumer container, the producer service also constructs a proxy service for representing the remote consumer. This proxy (C') is on charge of redistributing primitive notifications to the remote consumers listening to the same data transport address. For that, the consumer proxy keeps the transport address where it has to send data messages and the list of control address of the consumer containers.

6. It is important to notice that the consumer proxy keeps track of interested containers not services. This means that if an additional service (C2 in the figure) subscribes to the same service the data from the producer consumer to the consumer service (from SC1 to SC2 in the figure) is sent only one time, being the consumer container on charge of replicating the data to all its interested consumer services.

7. In fact, in the producer container only has one proxy for representing all the remote services listening to the same transport. If we start a new service container (SC3) and it subscribes to the producer service using the same transport address (i.e a multicast address or a UDP broadcast address), only a new control address will be added to remote consumer proxy but the same proxy and transport will be used thus minimizing the transmission costs.

8. At this point the publication-subscription establishment has finished and data from the primitive will flow from the publisher service to all the consumer services. P1 notifies all its local subscribers by calling its service callbacks (i.e VariableChanged() for variable primitives). The proxy implementation of this callback, will effectively encode the information and sent the data to its data address through the network.

9. The message containing the data arrives to all the interested containers (SC2). After being decoded, its payload data is replicated to all the consumer services (C1,C2) which receives the data through a service callback.

10. Publish-Subscribe relationship is terminated in an analogue way and it is not reflected for figure clarity.

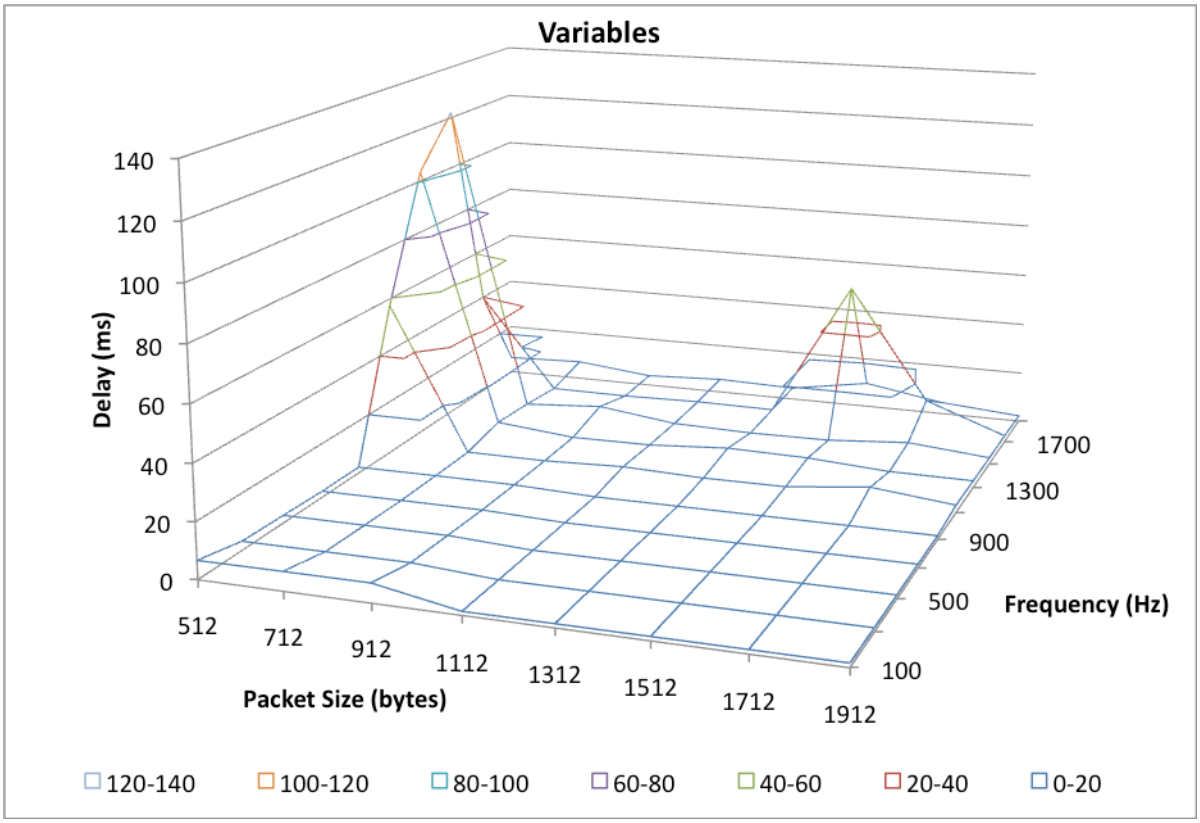

Figure 6. Marea Variables Performance. *The figure shows the performance of the Variable primitive for different packet sizes and frequencies.*

D. Marea performance tests

Marea has not been designed for hard real time applications (i.e the control loop of an autopilot). In our architectural view this real time requirements are contained inside black box services (i.e the autopilot). These specific services are implemented following hard real time constrains while the connection with the rest of the avionics system can comply with less constrained requirements. For example, in our platform we are using a UAVNavigation[9] commercial autopilot implemented as a monolithic application over a real time operating system (RTOS). This component is connected by a serial port to a Marea service, the Virtual Autopilot System (VAS), that acts as a driver to the other services in the platform. In the communication between the VAS and the service on charge of feeding the waypoints from the flight plan will suffice with soft real time requirements: low variance. On the other hand, if Marea has to be used for implementing complex interactions between services, low latencies and high bandwidths should be attained.

For this preliminary evaluation, the performance experiment consist on communicate two services deployed in two service containers connected by a local network. The first service starts a timer and sends a variable to the second service, this one returns the variable to the original producer. In that moment the timer is stopped for this variable and the round trip latency is calculated. In addition, lost packets and out of order packets are computed. In the figures 6 and 7, they are shown some performance metrics for the current implementation of Marea. The most two common primitives are used, variables and events respectively. The X axis represent the packet size in bytes, the Y axis represent the frequency of the communication, and finally, the Z axis represent the latency obtained.

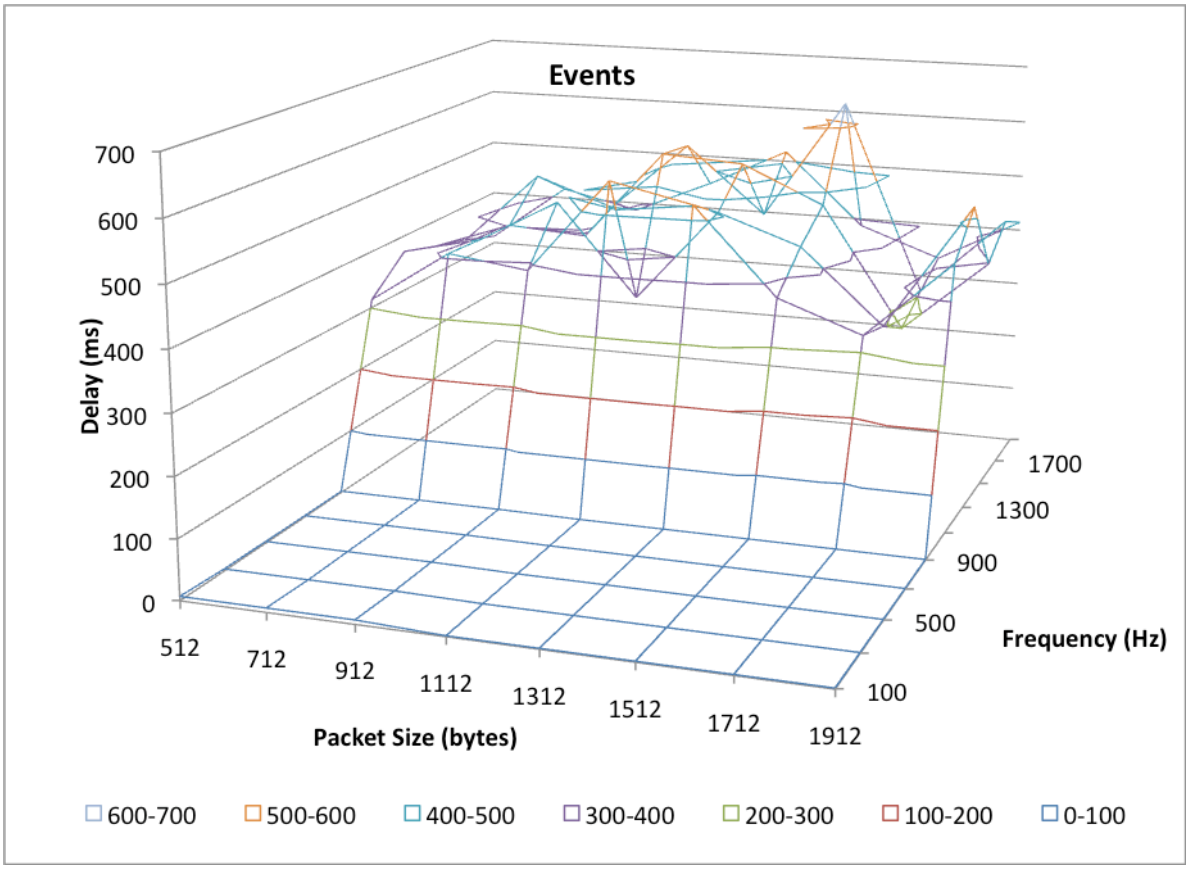

Figure 7. Marea Events Performance. *The figure shows the performance of the Event primitive for different packet sizes and frequencies.*

For this specific experiments two commercial-off-the-shelf (COTS) PC-based devices has been used. Next are the characteristics of the two nodes and the local area network:

- Node 1
    - CPU Intel Pentium 4 3.0 GHz
    - 3GB RAM
- Node 2
    - CPU Intel Pentium 4 3.0 GHz
    - 512 MB RAM
- Network
    - 100Mb LAN

The applicability space of the Marea middleware is determined by the area in which latency is low. As we can see in the graphics until 900Hz frequencies the results are while the message size (at least for the ranges tested) seems to not contribute in excess to the latency of the system. Higher frequencies saturate the system and results in inconsistent behavior. Variable primitive has little latency than not the Event primitive. This is consequent with the mapping to UDP datagrams and TCP streams. However it is important to note that, under heavy loads the Variable primitive will keep lower latency but will lose packets to keep this rate.

In this experiment only one-to-one communication has been tested, with several receivers the Variable primitive results will be the same (the Marea container will sent UDP datagrams in broadcast mode), however Event primitive keep growing linearly as the same data should be repeated for each consumer service.

For our requirements having frequencies less than 1000 Hz is enough for communication between mission avionics services. However, high-end hardware used for this preliminary experiment is not suitable to be embedded in a mid-range UAS. Future performance tests will be done with embedded boards that will generate more realistic results for the scenario.

III. Communication Gateway

In a UAV environment it is very common to use different links to support communication at different ranges and to provide redundancy in the case a link is down[1]. These point-to-point links usually do not support multicast and may have associated costs (economical or power-consumption) that restrict their usage to specific situations or very important transmissions[1]. The UAV should be intelligent enough to send the data through the most appropriate channel. This decision should be based on the type and length of the data to send, the current quality of the different links and the mission status. At the same time, the communication between the UAV and the ground station should be as much uniform and transparent to the services as possible in order to allow the deployment of the same services in different missions over different network relays.

Marea has a specific system called Communication Gateway that it is on charge of managing air-to-air and air-to-ground UAS communications. Their main purpose is to make the network transparent to the different services that use it. Our middleware assumes a high-speed Ethernet-like network that interconnects the different service containers. This local area network provides a very good bandwidth/cost ratio and very high reliability. In addition, in Ethernet networks broadcast and multicast communications are straight-forward and low-cost operations. The underlying protocol of our middleware heavily relies on these characteristics. However these characteristics cannot be guaranteed when other types of network links are used, for example radio frequency modems, sat-links or 802.11 wireless networks.

A. Communication Gateway requirements

The basic requirements that the Communication Gateway has to fulfill are:

1) **Support for several datalinks**
   The UAS will have several datalinks that the services should use in command and transparent manner. New datalinks should be easily configured to be used.

2) **Network hardware and protocol independence**
   The services should not be modified because using one network technology or another. The underlying network can use IP or not. For example, most RF modems provide a wireless RS-232 link; this means no data transmission integrity, no error correction and no retransmissions. Marea Transport component

implementation should resolve this transport-specific problematic while encapsulating and hiding these details to the rest of the system.

3) **Transparent handover between networks.**
When a communication session has been established between two remote service containers changing from a datalink to other should be transparent to the services involved. Imagine that we are using a high-throughput 802.11a WiFi connection to while the UAS is near the ground station, as the distance increases, signal will decrease and the UAS Communication Gateway will change to RF modem communication. Established sessions will hand over from a datalink to the other.

4) **Prioritize traffic from different services.**
When the available bandwidth is not enough for all the required data flows, the Communication Gateway should prioritize the traffic from specific services, i.e the autopilot. Other traffic can be discarded, postponed its transmission, or stored in persistent medium for later processing on ground. For, example command data should be prioritized always over telemetry data.

5) **Using simultaneously several networks for higher throughputs**
If a UAS configuration provides several datalinks and in a specific point of the mission there is a high throughput necessity and they are available, Marea container should distribute the generated data through the different datalinks. Imagine that we have two RF modem on different bands and we want to download to ground a high-resolution aerial photo, by using the two RF modems simultaneously we will be able to finish the job in half the time.

6) **Security and encryption**
Marea protocol has some security checks, but in general it has not need to cope with malicious services, because the internal local network can be considered safe. When the communications are done external to this network and using a wireless channel the possibilities that this channel can be attacked are highly increased. Some effort is done in this area, although the current implementation of the current gateway lacks these features.

B. System architecture

The idea of the communication gateway is to add a new layer on the stack that will manage the multiple datalinks and decide which to use for each data flow. This process is dynamic as it depends on the links status. Link status can change from available to unavailable in any moment of the UAS mission, on that case, the Communication Gateway will be on charge to transparently handover the communications from one link to other. One advantage of separating this functionality on an independent layer is that the service containers present on internal networks not needing handover and multiple network interface functionalities can remove completely this layer avoiding all the resources cost.

Communication Gateway layer is installed between the Protocol and Encoding layers, just before deciding which encoding use. It only processes output messages. Input messages are bypassed and their differentiation is performed at the scheduling done by the Protocol layer. Messages from high priority services are queued before.

An XML configuration file defines which transports to use and which policies to apply to each sort of packets. The packets are mainly classified by their producer and their consumer. In addition to this static information, the Communication Gateway receives dynamic information from the Transport layer, mainly the link status of all the transports enabled. The Transport components on the transport layer has been extended to be able to provide this information.

All information related to link status and interfaces are stored in a component called Network Manager. The Network Manager it is on charge on relating addresses with its link status. With this both dynamic and static information the Selector component decides to which interface send each Message.

Services and service container layers use specific objects called TransportAddress to identify the recipient of a Message. This object not only carries the address of the underlying specific networks (for example its IP address) but also information about the encoding to use. This TransportAddress will be used by the lower layers to select the appropriate encoding and interface. Communication Gateway only needs to route the Message through the lower layers by changing the TransportAddress of the Message.

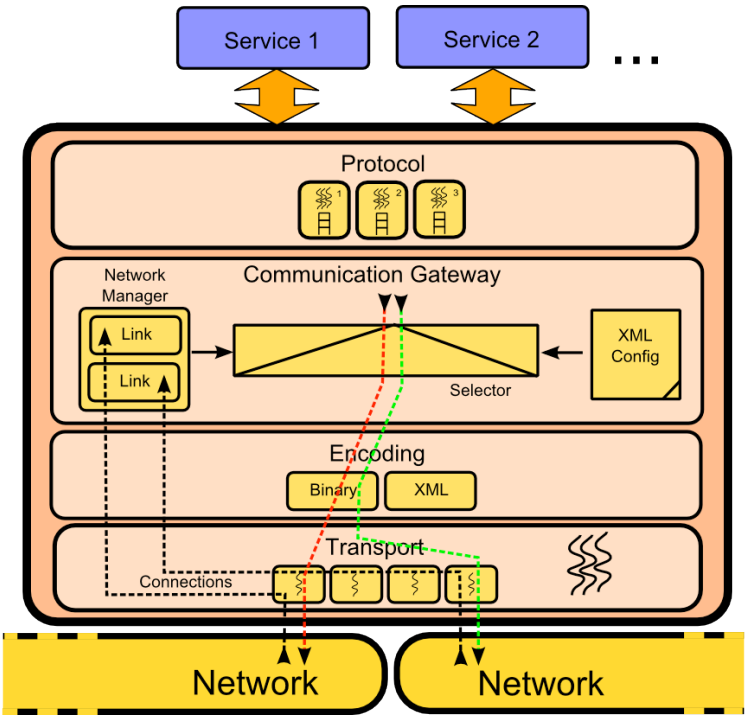

Figure 8. Communication Gateway architecture. *The figure shows the layers of the Marea middleware interacting with Communication Gateway.*

A general view of the architecture is shown in figure 8. After being processed by the Protocol layer, two packets arrive at the Communication Gateway. With the information gathered from the Transport layer and stored in the Network Manager about the two Link available the Selector decides which route to use. The XML configuration files defines that the Messages from Service 1 are encoded in binary and sent through network 1 (Wi-Fi), while the Messages from Service 2 are encoded also in binary, but sent through network 2 (RF).

In addition to the Communication Gateway layer, a specific SerialTransport has been implemented to be able to use Radio Frequency Modems based on RS-232 interaces, for example the Digi 2.4Ghz XStream module.

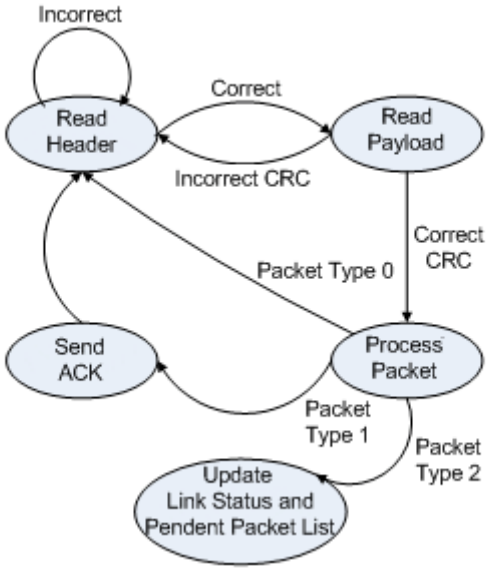

Figure 9. Serial RS-232 Communication States. *The figure shows the state diagram implemented for Serial RS-233 communications.*

RS-232 communications lack algorithms or protocols for error detection and correction so this features have to be implemented in the Transport layer. In figure 9, it is shown state machine that has been implemented for CRC checking and automatic retransmission of lost packets.

C. Protocol

Two changes has been done in the Marea protocol to adapt it the communication gateway requirements. First, a mechanism to discover and autoconfigure the relationship between links and service container addresses. Second, the proxy mechanism has been extended with a specific proxy to manage the services deployed on services containers accessible via the Communication Gateway links.

The Communications Gateway bases its operation in knowing which other service containers with the Communications Gateway layer enabled are in network range. In order to do so, a discovery-response mechanism has been used. This behavior is dynamic, taking into account that not always the network links are available and with the same peers at the other end. Therefore, a table relating service containers with the Communication Gateway layer and the various links that can be used to communicate with them is obtained and stored in the Network Manager.

Regarding services communication, all the messages involving the discovery, publication, subscription and notification of services are forwarded through the available remote links, changing the control reply-to address to itself. In other words, the gateway acts as an intermediary for services in all the service containers of its local network. For managing this intermediation, the Communication Gateway creates specific proxies: remote producers and remote consumers. The difference with original producers and consumers is that they forward it to next gateway instead of using the payload data contained in the message.

D. Configuration

The configuration of the Communication Gateway is stored in the file *gateway.xml* and it contains all the static information needed to perform its task. If the file is not present the system assumes that the container is deployed in a local network and all the Communication Gateway Layer is disabled.

```
<transports>
  <ip name="eth" interface="eth0"/>
  <ip name="wifi" interface="wi0" dst="10.0.0.2"/>
  <serial name="rf1" interface="COM4" rto="3000" dst="10.0.0.2"/>
  <serial name="rf2" interface="COM5" rto="3000" remote no-default/>
</transport>
```

Figure 10. Communication Gateway Configuration (I). *The figure shows the transport section of the gateway.xml configuration file.*

In figure 10, it can be seen four different interfaces configured. The first one is the local Ethernet, the second one is a Wi-Fi network which gives access to another service container that is bound to the IP address 10.0.0.2. In this configuration we can see how a static route is defined for the configuration gateway.The two later interfaces are RF modems accessed by the RS-232 serial transport. The *rf1* serial transport is configured statically to the same service container bound to IP address 10.0.0.2. On the other hand, the *rf2* serial transport uses the gateway discovery mechanism to locate the service containers on the other side of the RF modem.

The policies section establishes different policies for each of the services that can send data through the Communication Gateway. The most basic form of selecting is by using regular expressions to select the source services. In figure 11, it can be seen three different policies: *autopilot-traffic*, *telemetry-traffic* and *default*. For each policy "from" clauses can be added to check if the packet arrives from the specified services. If any of the expressions matches then the packet is sent through the specified lane. For example for autopilot traffic all the messages are sent through the lane *autopilot*. Telemetry traffic both from services TemperatureSensor and Humidity sensor are sent through the lane *telemetry*. And finally, the rest of messages are sent through the lane *default* as is expressed in the last clause of the XML.

```
<policies>
  <policy name="autopilot-traffic" lane="autopilot">
      <from address="/*/*/VirtualAutopilotSystem/*/*"/>
  </policy>
  <policy name="telemetry-traffic" lane="telemtry">
    <from address="/*/*/TemperatureSensor/*/*"/>
    <from address="/*/*/HumiditySensor/*/*"/>
  </policy>
  <default lane="default"/>
</policies>
```

Figure 11. Communication Gateway Configuration (II). *The figure shows the policies section of the gateway.xml configuration file.*

A lane programmatically defines the course that the messages assigned to it will follow. It supports several clauses to process the messages. The most used ones are *route* and *store*: *route* basically sends the packet through the specified data link, and *store* keeps the payload data of the messages in a persistent storage media for later processing. If a lane his configured with different clauses, they are tested one by one in order and the first that can be used is executed, finishing the processing of this message.

For example in figure 12, *telemetry* lane tries to sent data through the WiFi interface, later through the first radio-modem and at last, if no communication is possible through these datalinks, the payload is stored time-stamped in a flight-log. *Autopilot* lane tries to use all the available links giving priority to RF modems *rf2* and *rf1* over the WiFi link. The *default* lane only routes the packet without any specific priority through an available interface that links with the destination of the message. It is important no notice that *rf2* link has been marked to be only used if explicitly called, and then it will not be used for this default configuration, keeping this channel reserved for autopilot messages.

```
<lanes>
  <lane name="telemetry">
    <route transport="wifi"/>
    <route transport="rf1"/>
    <store name="flight-log" timestap="true"/>
  </lane>
  <lane name="autopilot">
    <route transport="rf2"/>
    <route transport="rf1"/>
    <route transport="wifi"/>
  </lane>
  <lane name="default">
    <route remote/>
  </lane>
</lanes>
```

Figure 12. Communication Gateway Configuration (III). *The figure shows the lanes section of the gateway.xml configuration file.*

IV. Conclusions

This paper present the implementation details of Marea, a middleware for UAS avionics. Marea provides a full distributed paradigm to the avionics designers. Each avionics semantic is encapsulated into a service, or piece of software which realizes a function. Data communication between services are always conducted through the middleware. The physical location of the services is transparent to the designers, being Marea responsible of the service discovery, location and execution.

Each distributed hardware element executes an instance of Marea, named Service Container. Avionics services are executed within the Service Container as internal threads. The software architecture of a Service Container is layer oriented: The services interface is in the highest layer, while the hardware interfaces are in the lowest.

The main targets in the Marea development process have been the communications efficiency, the transparent service location and the automatic selection of the physical link for the communications. The communications efficiency is defined as the relation between the transfer speed and system cost. Marea defines four different communication primitives. The primitive type defines the criticism of the data. The location of source and recipient services of the message define the best link for the data transfer. Marea implements a protocol to best schedule each

message. The protocol is the highest layer of the Service Container. Performance measures demonstrate that the approach is robust for the mission avionics we target.

The paper presents also the second layer, the gateway. This layer has information from the lowest layer about all the available links and their status. The gateway, using a configuration file, decides the best physical channel. Service ubiquity is the final result for avionics designers.

As future work we are defining new performance tests for a large number of services, including also Remote Invocations and File Transfers primitives. Also a dynamic decision algorithm can be implemented for the gateway and measures with different channels have still to be obtained.

Acknowledgments

This work has been partially funded by Ministry of Science and Education of Spain under contract CICYT TIN 2007-63927. The work presented in this paper has been performed with support from the Innovative Studies Programme of the EUROCONTROL Experiment Center.

References

[1]Richard S.Stansbury, Manan A.Vyas, Timothy A.Wilson, "Survey A of UAS Technologies for Command, Control, and Communication (C3)" *Journal of Intelligent Robot Systems (2009)* 54:61–78

[2]Harold Carr, "Server-side Encoding, Protocol and Transport Extensibility for Remote Systems" *ICSOC'04,* November 15–19, 2004, New York, New York, USA.

[3]M. Rose, "The Blocks Extensible Exchange Protocol Core" *W3C - RFC-3080,* March 2001

[4]J.López, P.Royo, C.Barrado and E.Pastor, "Modular Avionics for Seamless Reconfigurable UAS Missions", *IEEE/AIAA 27th Digital Avionics System Conference DASC'08,* October 2008, St.Paul, Minnesota, USA.

[5]Venkita Subramonian et al, "Fine-grained Middleware Composition for the Boeing NEST OEP", *OMG Workshop on Real-time and Embedded Distributed Object Systems,* July 2002.

[6]J.López, P.Royo, C.Barrado and E.Pastor, "A Middleware Architecture for Unmanned Aircraft Avionics", *Middleware'07,* November 2007, New Port Beach, California, USA.

[7]D.C.Schmidt, "Middleware for Real-time and Embedded Systems", *Communications of the ACM,* June 2002, 43-48.

[8]C.Honvault, M.Roy, P.Gula, J.C.Fabre, G.Lann and E.Bornschlegl, "Novel Generic Middleware Building Blocks for Dependable Modular Avionics Systems", *EDCC 2005 - LNCS,* 2005, 3463:140-153.

[9] "UAV Navigation AP04 autopilot" http://www.uavnavigation.com/uavprod/uavprod_01.htm

[10] "ARINC Specification 459" Published by ARINC, 2551 Riva Road, Annapolis, MD 21401. http://www.arinc.com

[11] "ARINC Specification 653. Avionics Application Software Standard Interface" Published by ARINC, 2551 Riva Road, Annapolis, MD 21401. http://www.arinc.com

[12] R.Garside, F.J.Pighetti, "Integratimg Modular Avionics: A New Role Emerges", *IEEE/AIAA 26th Digital Avionics System Conference DASC'07,* October 2007, Dallas, Texas, USA.